GEEDGE networks | Guarantee your edge network security.

编程的艺术

如何写出好代码

报告人：郑超

2020-8-13

目录

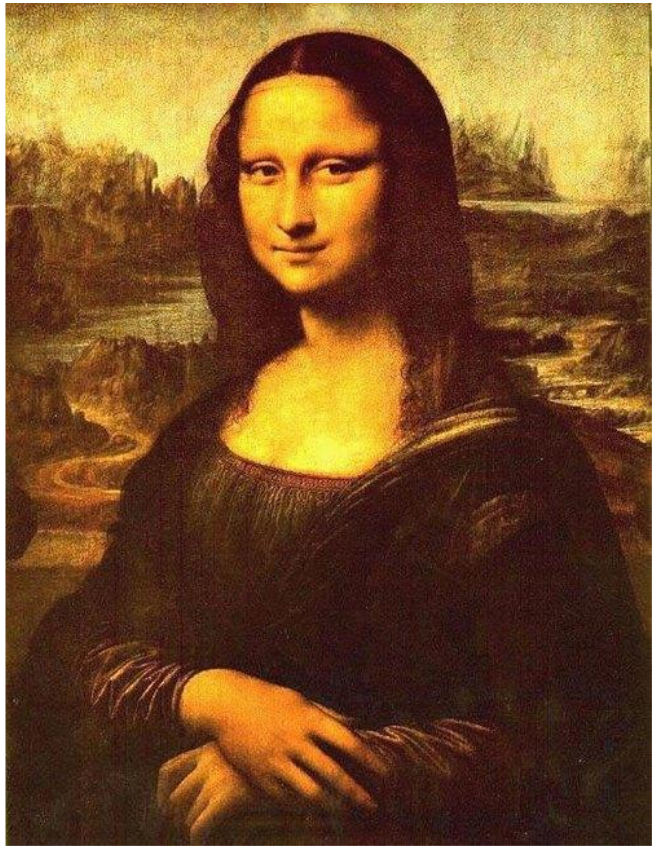
- 建立艺术品味
- 好代码始于得体的命名
- 分布式架构
- 编写代码
  - 程序->模块->函数->代码段
- 优化、调试与重构
- 不断的修炼

什么是好代码?

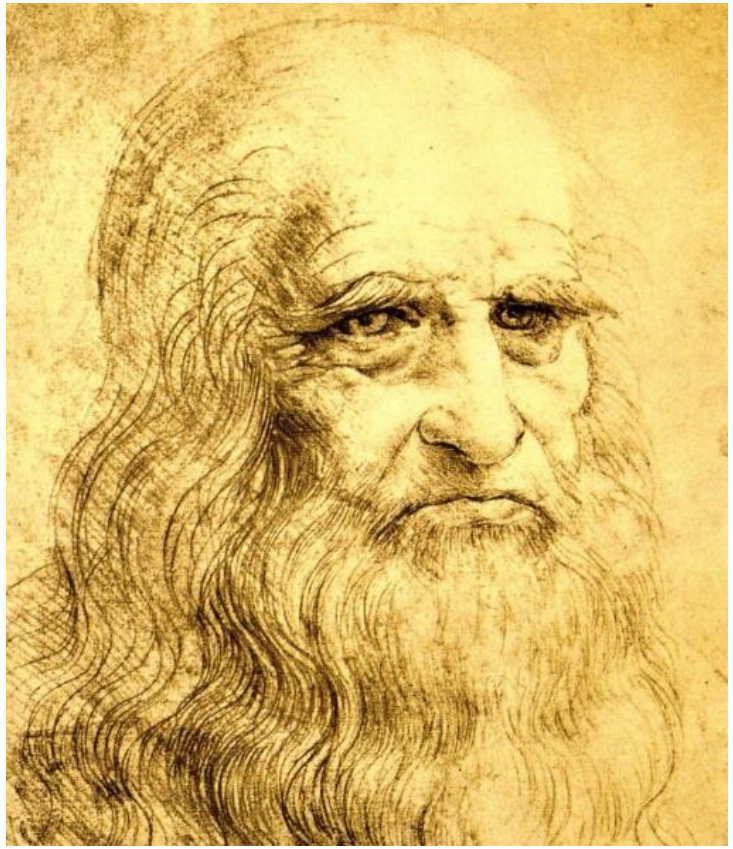
编码是一种艺术。

代码是人类智慧的结晶，反映了一个人/团队的精神!

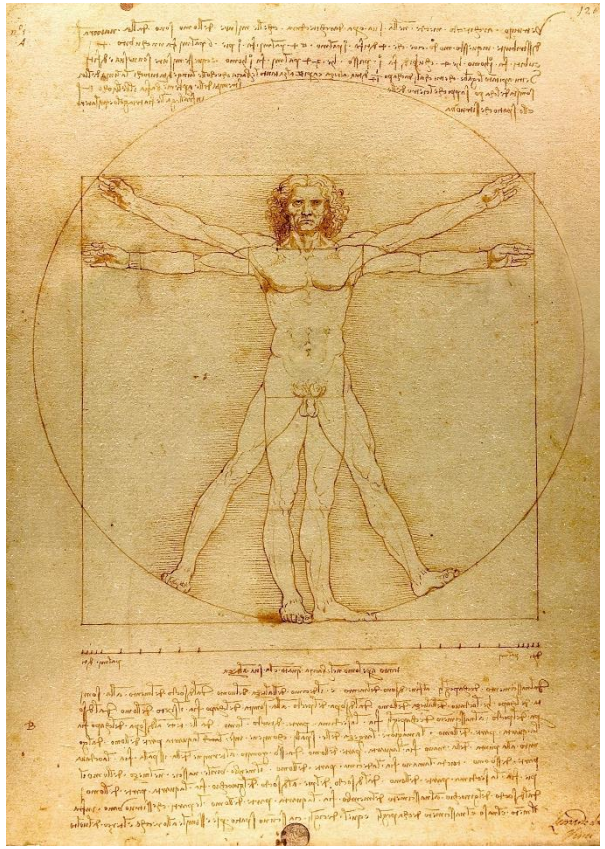
艺术，解读的角度



艺术品



艺术家



艺术家的思考方式

好代码来自优秀的软件工程师

- 软件工程师 != 码农
- 软件工程师应该具有综合的素质
- 技术
  - 编码能力，数据结构，算法
  - 系统结构，操作系统，计算机网络，分布式系统
  - 英语，文献阅读，社区讨论
- 产品
  - 对业务的理解，交互设计，产品数据统计，产品/业务运营
- 项目管理
- 研究和创新
  - R&D: Research and Development
- 一个合格的软件工程师的成长需要8-10年的时间

Linus Torvalds

Jeff Dean

一流代码的特性

- 可读性
  - 简洁、简短
  - 变量、函数的命名合理，他人不用思考就能看明白
- 可测试
  - 很容易进行单元测试
  - If it's hard to document, it's probably wrong. If it's hard to test, it's almost certainly wrong.
- 可重用
  - 功能的定义是经过抽象和概括的
- 可维护
  - 状态可监控，可运维

引用自：

《写好代码的十个秘诀》，林斌，2000；

《On Designing and Deploying Internet-Scale Services》, Windows Live Services Platform, 2007

《代码的艺术》，章淼，百度技术学院课程，2016

坏代码的一些表现

- Bad name
  - Function Name: do(), myFunc(), …
  - Variable Name: 非循环变量的a, b, c, i, j, k, temp
  - 莫名其妙的数值
    - if number>11
- No abstraction
  - 缺乏"讲故事"的能力
- The function has no single purpose
  - LoadFromFileAndCalculate()
- Bad layout
  - 多种风格混搭
  - 相同的事务有多个别名
- 重复/相似的代码

目录

- 建立好代码的品味
- **好代码始于得体的命名**
- 编写代码
  - 程序->模块->函数->代码段
- 优化、调试与重构
- 分布式架构
- 不断的修炼

好的代码从哪里来?

- 代码不只是”写”出来的
  - 编码的时间一般只占10%
- 好的代码是多个环节工作的结果
- 编码前
  - 需求分析，系统设计
- 编码中
  - 编写代码，单元测试
- 编码后
  - 集成测试，上线，持续运营/迭代改进/重构
- 一个好的系统/产品是以上过程持续循环的结果

Coding并不容易

- Coding是从无序变为有序
  - 将现实世界中的问题转化为数字世界的模型
  - 一个认识的过程(从未知变为已知)
- 在Coding的过程中，需要
  - 把握问题 的能力
  - 建立模型 的能力
  - 沟通协作 的能力
  - 编码执行 的能力
- 写好代码需要首先建立品味

需求分析 & 系统设计

- 这是两个经常被忽视的环节
  - 很多人错误的认为，写代码才是最重要的事情
  - 在没有搞清楚目标之前，就已经开始编码了
  - 在代码基本编写完成后，才发现设计思路是有问题的
- 软件开发规律和人的直觉是相反的
  - 在前期更多的投入，收益往往最大
  - Why？除了开发，测试、上线、调试等都是很大的成本
- 二者的区别
  - 需求分析：定义系统/软件黑盒的行为 (external, what)
  - 系统设计：设计系统/软件白盒的机制 (internal, how& why)

需求分析

- 问题：怎么用寥寥数语勾勒出一个系统的功能?
- 每个系统都有自己的定位，以Google File System (GFS)为例
- GFS is designed to provide efficient, reliable access to data using large clusters of commodity hardware.
- 怎么描述GFS的需求?
  - 分布式文件系统
  - 文件数量；文件大小的分布；总的存储容量
  - 读写能力（读写文件次数，数据传输速率，读写延迟）
  - 容错的能力；一致性方面的定位（强一致，弱一致）
  - 对外的接口（用户怎么使用）
  - …
- 用精确的数字来刻画需求
  - 量变导致质变

系统设计

- 首先是对外接口的设计，比系统实现本身还要更重要
  - 接口定义了**功能**：如果功能不正确，系统就没有用
  - 接口决定了**外部关系**：相对于内部，外部关系确定后非常难以修改
- 哪些是接口？
  - 模块对外的函数接口，如：动态链接库
  - 平台对外的API，如：微服务
  - 系统间通信的协议
  - 系统间存在依赖的数据
    - 如：给另外一个系统提供的词表
- 设计和修改接口，需要考虑的非常清楚
  - 合理，好用
  - 修改时需要尽量向前兼容

*There are only two hard things in Computer Science: cache invalidation and **naming things**.*

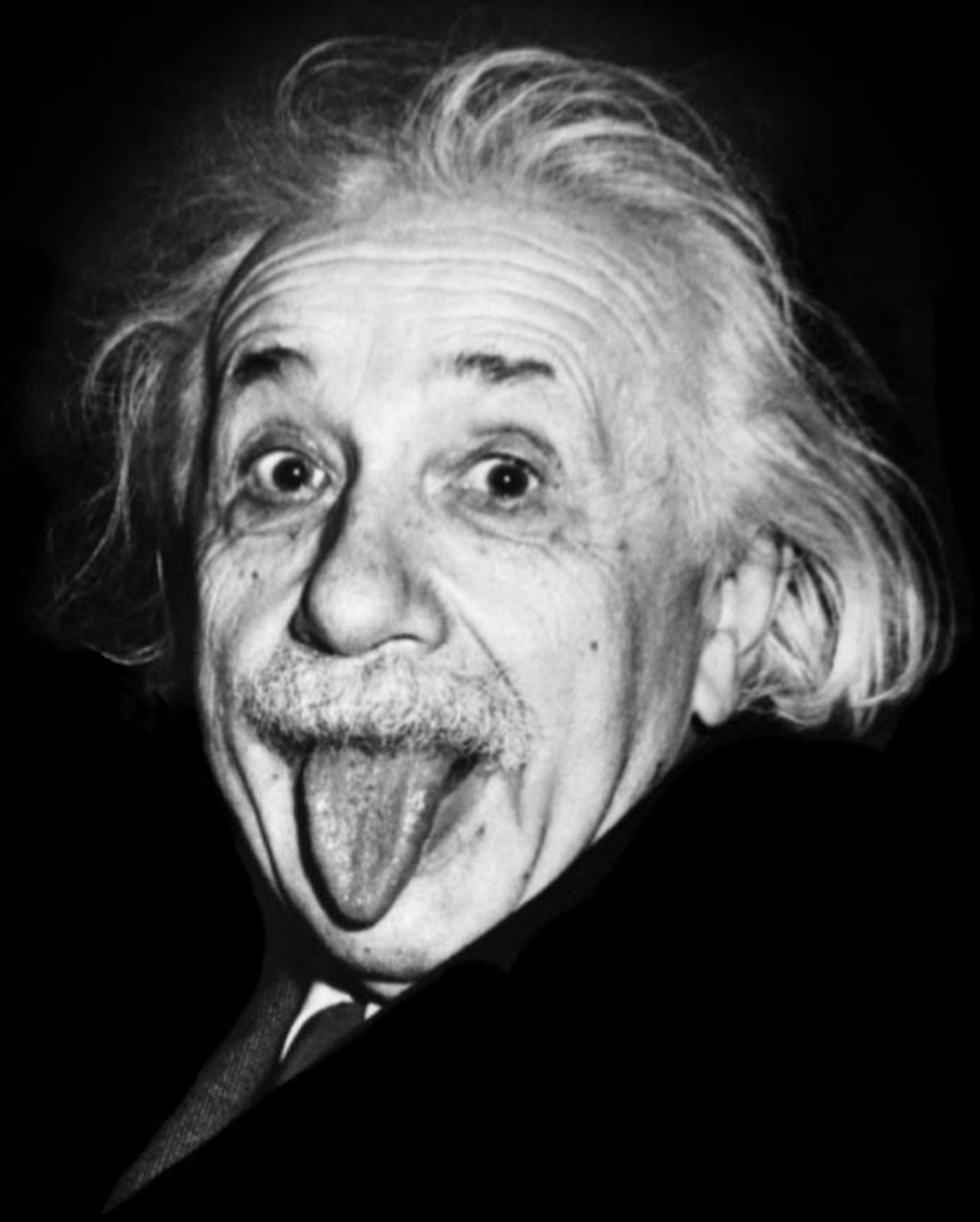
-- Phil Karlton

命名驱动设计

- 好代码源自得体的命名
- 名字决定格调
  - 王语嫣->王云
- 例如
  - Data re-range -> Range Rover
  - IP Discovery -> Web Focus
  - Cyber Narrator (网络叙述者)
  - App Sketch (App速写)
  - Sanity Directory (心智目录)
- 它是相互交流的开发代号
  - 会作为代码文件名、函数名、变量名在你的源代码中反复出现
  - 上线后还会成为可执行文件名。

Naming Objects

"If you can't explain it simply, you don't understand it well enough."



Naming Objects

- 识别系统中的各种数据
  - 数据压倒一切。
  - 如果已经选择了正确的数据结构并把一切都组织得井井有条，正确的算法也就不言自明。

编程的核心是数据结构，而不是算法。

——Rob Pike，Notes On C Programming

- 命名的原则
  - 沿用已有术语，不要生编硬造
  - 寻找术语的过程，就是在调研前人解决方案的过程

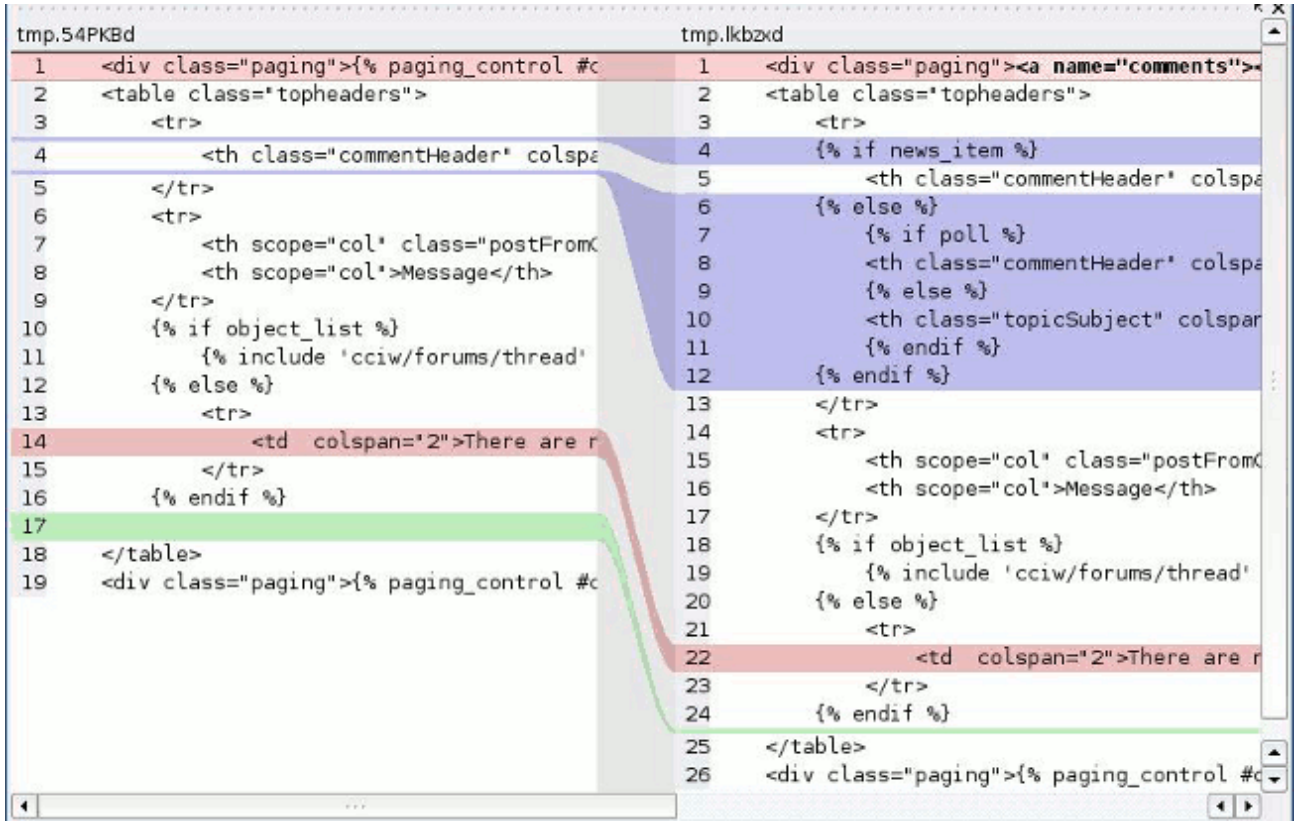
太阳底下没有新鲜事儿。

——《圣经·旧约·传道书》

- 命名的过程就是建模的过程

举例：比较两个文本的差异——*diff*

- 什么叫文本?
  - 一个连续的字符串
- 什么叫差异?
  - A变化到B的最小集
- 怎么展示差异?
  - 增、删、改
- 问题的本质
  - 编辑距离?
  - 最长公共子串?
  - 最长公共子序列



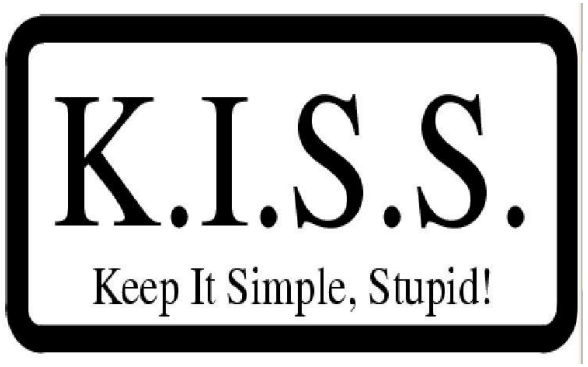
系统设计

- 什么是系统架构(System Architecture)?
  - A system architecture is the conceptual model that defines the structure, behavior, and more views of a system.(from wiki)
- 几个要素
  - 系统要完成哪些功能
  - 系统如何组成
  - 功能在这些组成部分之间如何划分
- 系统设计的约束
  - 资源的限制：计算，存储，IO/网络
- 需求是系统设计决策的来源
  - 在设计中，经常需要做Trade-Off
  - 需求是make decision的重要依据

系统设计的风格和哲学

- 在同样的需求下，可能出现不同的设计
- 电信网络 vs. Internet
  - 中心控制 vs 分布式控制
  - 可靠组件 vs 不可靠组件
- CISC(复杂指令集) vs RISC(精简指令集)
- 好的系统是在合适假设下的精准平衡
  - 一个通用的系统，在某个方面是不如专用系统的
- 每个组件（子系统/模块）的功能都应该足够的专注和单一
  - 功能的单一是复用和扩展的基础
- 子系统/模块之间的关系应该简单而清晰
  - 软件中最复杂的是耦合（为什么全局变量是要极力避免的？）

设计是对复杂性管理

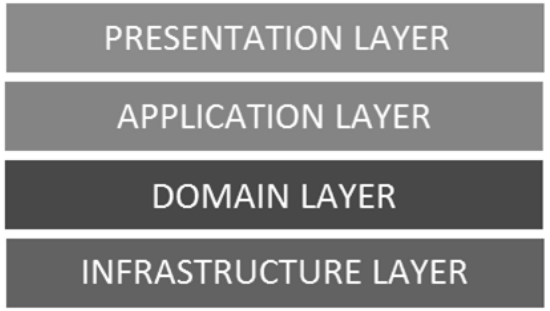


软件设计有两种方式：一种是设计得极为简洁，没有看得到的缺陷；另一种是设计的极为复杂，有缺陷也看不出来。第一种方式的难度要大得多。

——C. A. R. Hoare, CACM Feb 1981

低耦合，高内聚

把问题空间分割为规模更小且易于处理的若干子问题，分割得越合理、越易于理解，在装配越顺利。

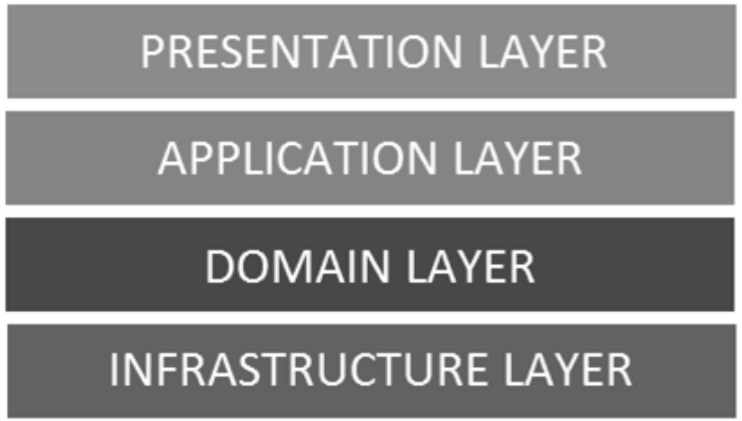


计算机科学中没什么问题是加个抽象层不能解决的，如果有，那就加两层。

——David Wheeler

经典分层：领域驱动设计

- 展示层
  - 请求应用层，获取用户所需的展示数据
  - 发送命令给应用层，执行用户的命令
- 应用层
  - 薄薄的一层，定义软件要完成的任务。
  - 对外为展示层提供各种应用功能，对内调用领域层（领域对象或领域服务）完成各种业务逻辑。
  - 应用层不包含业务逻辑
- 领域层
  - 表达业务概念、业务状态信息及业务规则，是业务软件的核心
- 基础设施层
  - 为其他层提供通用的技术能力，提供了层间通信；为领域层提供持久化机制。



目录

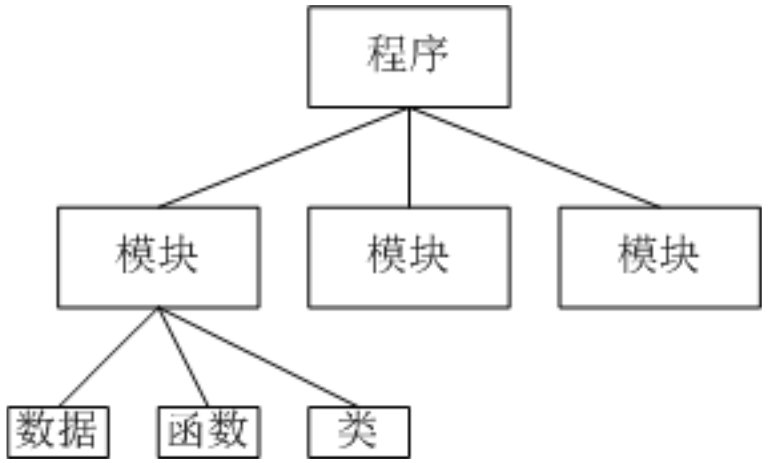
- 建立好代码的品味
- 好代码始于得体的命名
- 分布式架构
- **编写代码**
  - 程序->模块->函数->代码段
- 优化、调试与重构
- 不断的修炼

程序/模块

- 输入和输出是什么?
  - 通信协议
    - HTTP、grpc
  - 序列化方式
    - json、avro、Proto buffer、xml
- 单线程or多线程?
- 使用哪种数据结构?
  - 队列、哈希表
- 上线吗? 预期性能?
- 有哪些配置参数?
  - 监听端口、线程数、缓冲区数

模块

- 模块是程序的基本组成单位
  - 一个.c/.py/.go文件就是一个模块
- 模块需要有明确的功能，需要慎重考虑
  - 同样遵循：高内聚，低耦合
- 切分模块的一种角度



- 数据类模块主要完成对数据的封装
  - 模块内部变量
  - 类的内部变量
- 对外提供明确的数据访问接口
  - 不暴露数据结构和算法
- 数据结构是编程的核心

数据类的模块

- 过程类模块
- 本身不含数据
- 调用“数据类模块”或“过程类模块”

过程类的模块

写代码的注意事项

- 避免代码的重复
  - 5行以上的代码段出现超过1次，就应该封装了
  - 代码的重复意味着：
    - 错误的重复
    - 糟糕的设计
- 单线程or多线程？
  - 不要用多线程
  - 除非CPU是性能瓶颈
- 数据和业务分离

> Threads are for people who can't program state machines.
>
> ——Alan Cox

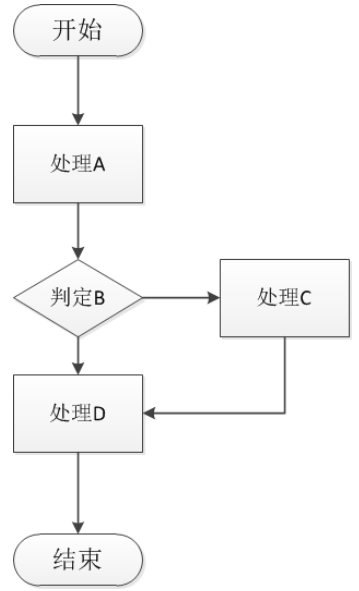
```
struct info
{
    int type;
    char* data;
    int len;
    struct info*nxt;
}
```

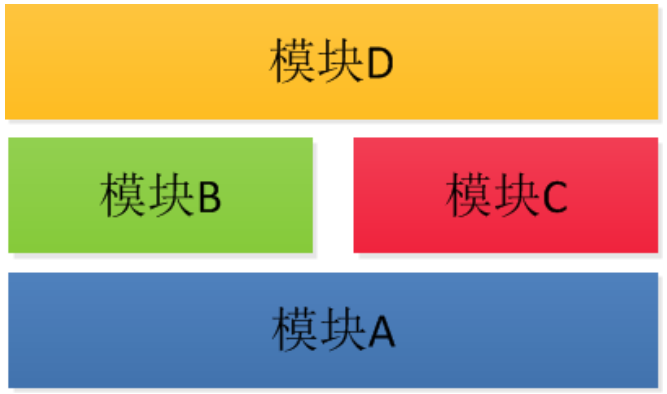
程序的透明性

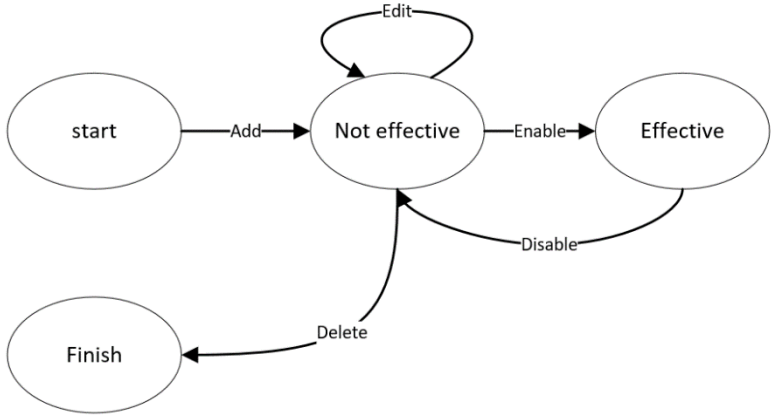
- 配置项、可能人工查看的中间结果要文本化
- 程序要能自证清白——易于测试的代码
  - 可测试：输入、输出均有记录
  - 可监控：展示统计功能，包括输入、缓存、输出的总量和瞬时速度等
- 程序要对运维人员友好
  - 配置项简明清晰，用户口算的参数
  - 输出的日志要有意义

必要时，你需要一个文档

- 超过500行的代码，你需要写一个文档记录你的开发过程
  - 结构框图
  - 重要函数或过程的流程图
  - 思维导图
  - UML







Right way vs. Quick way

- 一个需求，由两个技术方案可以满足：
  - 方案一：抽象合理，设计完备，开发周期半年
  - 方案二：直奔主题，简单粗暴，开发周期两周
- *灭火的故事
  - **火在咆哮**：最后期限、发布日期、会展演示等等

消防员在前门和着火出之间铺上垫子，因为他们不想弄脏地毯。

Quick way的代价：技术债务

Technical Debt Quadrants

	Reckless	Prudent
Deliberate	*"We don't have time for design"*	*"We must ship now and deal with the consequences"*
Accidental	*"What's Layering?"*	*"Now we know how we should have done it"*

*引自《The Pragmatic Programmer: From Journeyman to Master》

源码文件的组织

```
├project root
├src/                          //源代码目录
│   ├modules/                  //子模块
│   ├inc/                      //依赖库的头文件
│   ├test/                     //测试代码
│   └.h、.c、.cpp              //编写的源文件
├vendors/                      //第三方库
├tools/
├bin/                          //或 run/
│   ├log/            //运行日志目录
│   ├conf/                     //配置文件目录
│   ├data/                     //输出文件目录（可选）
│   ├worker_exe                //可执行文件
│   └guard.sh                  //守护脚本
├Makefile/Cmakelist.txt
└Readme.md                     //说明文档
```

源码目录结构

函数

- 系统=> 子系统/程序=> 模块=> 函数
  - 函数的切分也很重要
- 函数描述的3要素
  - 功能描述：这个函数是做什么的
  - 传入参数描述：含义，限制条件
  - 返回值描述：各种可能性
- 函数的规模
  - 要足够的短小
  - Python：尽量在一屏内（约30行）完成
  - C/C++：尽量在两屏内完成
- 写好程序的一个秘诀：把函数写的短一些
  - Bug往往出现在哪些非常长的函数里
  - 出现危险的根源，往往是过于自信

函数的返回值

- 每个函数应该有足够明确的语义
  - 基于函数的语义，函数返回值有***3种类型***
- True, False
  - 逻辑判断型”的函数，表示“真”或“假”
  - 如：is_white_cat()
- OK, ERROR
  - 操作型”的函数，表示“成功”或“失败”
  - 如：data_delete()
- Data, None
  - “获取数据型”的函数，表示“有数据”或“无数据/ 获取数据失败”
  - 如：data_get()

代码块

- 讨论范围：一个函数内的代码实现
- 思路：把代码的段落分清楚
- 形式的背后是逻辑（划分，层次）
  - 不要认为那些空行/ 空格是可有可无的
- Don’t make me think
  - 一眼看过去，如果无法看清逻辑，这不是好代码
  - 好的代码不需要你思考太多
  - 记住：代码更是写给别人看的
- 注释不是补出来的
  - 先写注释，后写代码
- 写代码就是讲故事
  - 一个在代码上表达不好的同学，在其他表达上一般也存在问题
  - 其他表达：文档、email、ppt、口头沟通、…

变量/函数命名

- 命名的范围：系统，子系统，模块，函数，变量/常量，...
- 为什么命名如此重要?
  - “望名生义” 是人的自然反应
  - 概念是建立模型的出发点（概念，逻辑推理=> 模型体系）
- 常见错误:
  - 名字不携带信息：do, a, b, ...
  - 名字携带的信息是错误的: 体会一下
  - set() vs update(), is_XXX() vs check()
- 命名不是一件很容易的事情
  - 要求：准确，易懂
  - 起一个好名字很多时候需要经过推敲
- 名字的可读性: 下划线，驼峰
  - ClassName, GLOBAL_CONSTANT_NAME, module_name

编写优美整洁的代码

- 必须遵循编码规范!
- 清晰胜于技巧，好的代码简单易读，像句子一样
- 函数和变量的命名、参数设计得体。
  - 过程函数的名字，可以用较强的动词带目标的形式，
    - 如check_order_info、is_valid_pkt;
  - 对于函数名字，可以使用返回值的描述
    - 如current_location;
  - 避免无意义或模棱两可的动词
    - 如process_input;
  - 名字太长也不好，可以用缩写，技巧是去掉元音字母
    - 如packet->pkt、count->cnt
  - 变量名为9～15个字母，函数15～20字母

减少出错的Tips

- 能用for循环就不用while循环
  - 自带变量初始化，易于复杂度评估
- 能用栈的内存就不要malloc
- 不要用If…else if..else if …else
  - 能不能用switch替代?
- 用数据而不是代码来描述业务逻辑
  - 表驱动方法
- 控制总代码量，行数越少意味你犯错的机会越少
- 早死早托生，当然要死个明白
  - 多用assert
- 出错处理
  - 需要判断Malloc的返回值吗?
- 编译时，Warning==Error

使用结构化数据

- 体现数据的联系
  - 容易找出哪一个数据是与另外一个数据相联的
- 简化成块数据的操作
  - 赋值
  - 交换
- 降低维护工作量
  - 增加或删除一个成员

分散的用户信息存储方案：

```
int user_id[100];
int phone[100];
int title[100];
```

结构化的用户信息存储方案：

```
struct user{
        int user_id;
        int phone;
        int title;
};
struct user my_company[100];
```

举例：结构化的用户信息

表驱动

- 用数据而不是代码来描述业务逻辑
  - 相比逻辑，数据不易出错
- 举例：返回每个月中天数(不考虑闰年)

```
IF Month=1 THEN Days=31
      ELSEIF Month=2  THEN Days=28
      ELSEIF Month=3  THEN Days=31
      ELSEIF Month=4  THEN Days=30
      ELSEIF Month=5  THEN Days=31
      ELSEIF Month=6  THEN Days=30
      ELSEIF Month=7  THEN Days=31
      ELSEIF Month=8  THEN Days=31
      ELSEIF Month=9  THEN Days=30
      ELSEIF Month=10 THEN Days=31
      ELSEIF Month=11 THEN Days=30
      ELSEIF Month=12 THEN Days=31
ENDIF
```

比较笨的方法

```
' INITIALIZE TABLE OF "Days Per Month" DATA
'
DATA 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    DIM DaysPerMonth(I)
    FOR I=1 TO 12
      READ DaysPerMonth(I)
    NEXT I

Days=DaysPerMonth(Month)
```

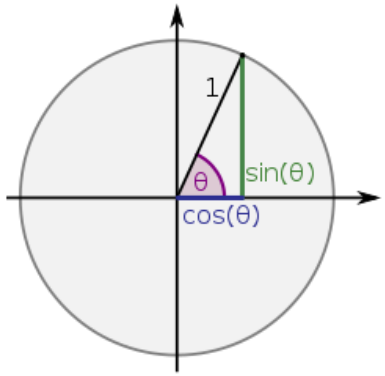
表驱动方法

目录

- 建立好代码的品味
- 好代码始于得体的命名
- 分布式架构
- 编写代码
  - 程序->模块->函数->代码段
- **优化、调试与重构**
- 不断的修炼

举例：计算三角函数

- 简单方案：直接调用Math库的sin函数计算
  - 时间复杂度$O(M(n)\log n)$，n为小数点保留位数



三角函数

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

泰勒级数展开

- 优化方案： 计算好0~2pi的值，存储在1024*128 的数组中，计算==查表
  - 时间复杂度O(1)

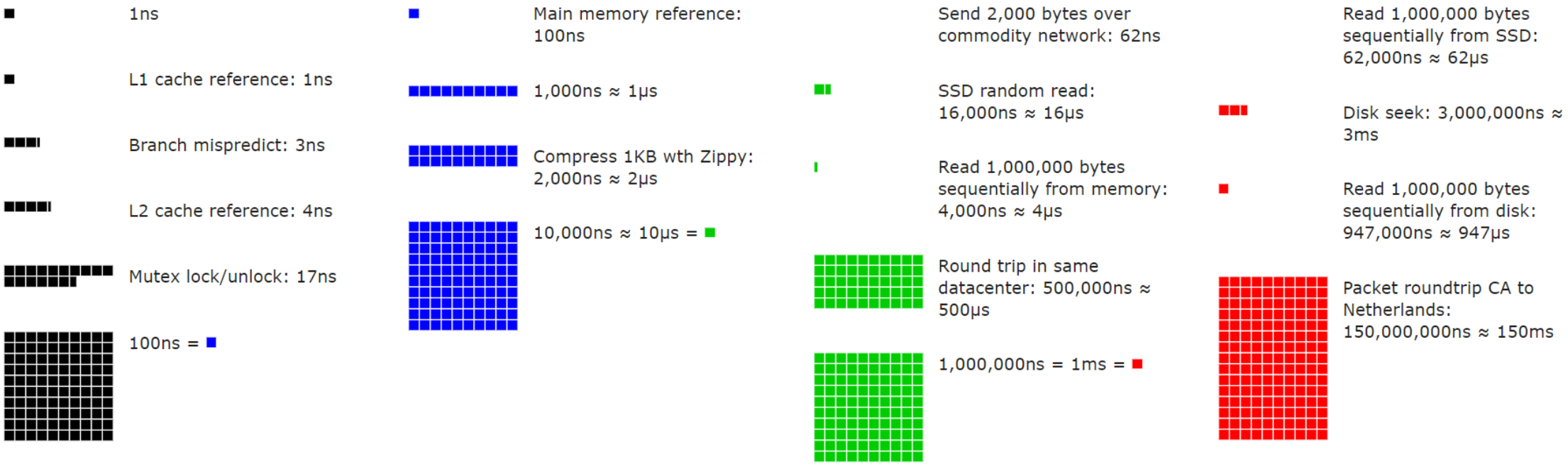
	简单方案	优化方案
-O0	6.2s	13.9s
-O3	5.2s	5.2s

*调用10亿次耗时

*CPU: Intel i7-8750H 4.2GHz 内存:DDR4 2666Hz

为什么优化没有效果?

- 查表需要访问内存
- CPU Cache让优化问题更复杂了
  - 随机访存是现代计算机的主要性能瓶颈之一



*Latency Numbers Every Programmer Should Know (2019)

* https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

什么时候优化?

- 瓶颈经常出现在想不到的地方，你根本不知道什么值得优化
  - 大型程序中，仅有3%的函数值得优化
- 花哨的算法通常会降低代码的可读性和提高复杂度，违背KISS原则
  - 花哨的算法在n很小时通常很慢，而n通常很小。
  - 内存池、线程池、连接池是新人的噩梦
  - 更难实现，更易出bug
- 现代编译器比你想象的强大
  - -O3

Make it work, make it right, make it fast.

—— Kent Beck

通常正确性第一，其次是简洁性，它可以为正确性提供更多保障；最后才应该考虑性能。

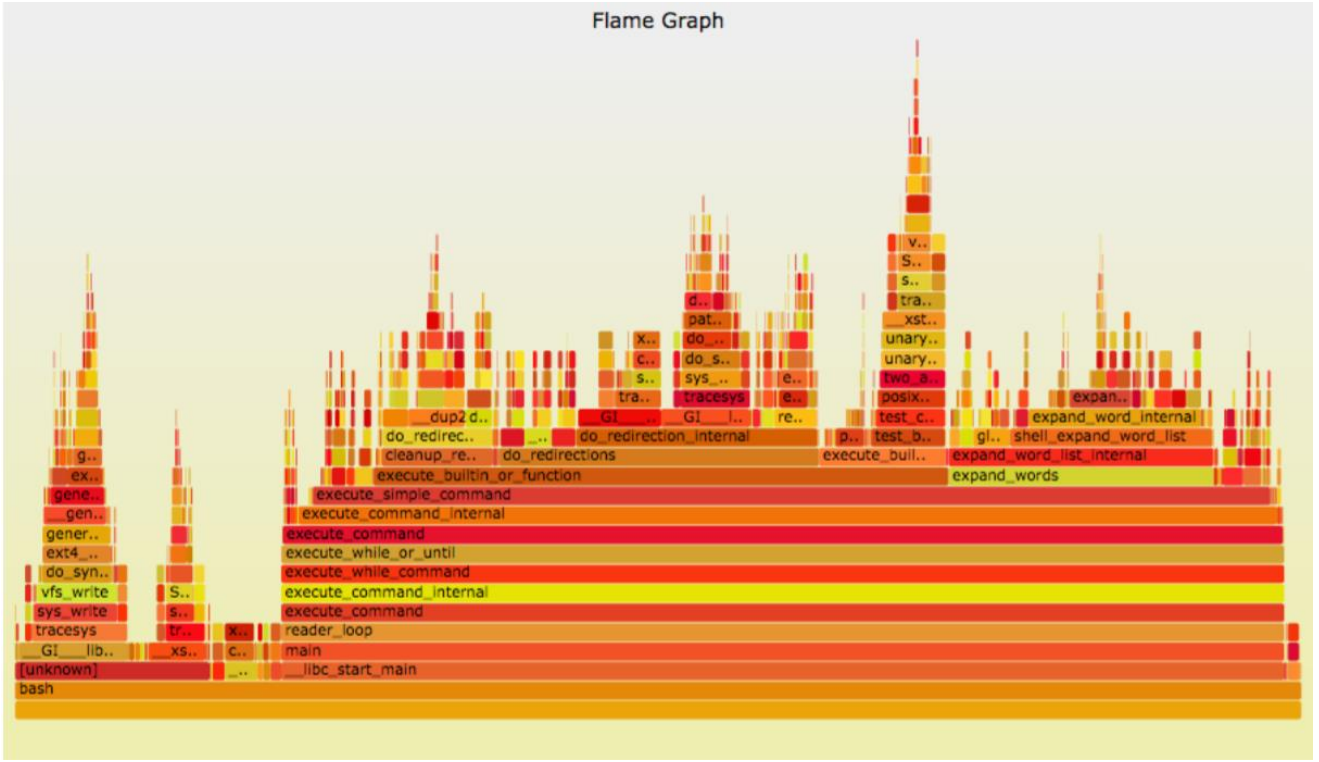
—— 云风

什么时候做性能优化?

- 用profile工具分析程序性能

```
Samples: 1K of event 'cycles', Event count (approx.): 353095948
Overhead  Shared Object           Symbol
  16.93%  libMESA_field_stat2.so  [.] get_bucket_index
  13.25%  libMESA_field_stat2.so  [.] lowest_equivalent_value
  10.47%  libMESA_field_stat2.so  [.] get_sub_bucket_index
   9.58%  libMESA_field_stat2.so  [.] move_next
   9.53%  libMESA_field_stat2.so  [.] hdr_size_of_equivalent_value_range
   8.73%  libMESA_field_stat2.so  [.] value_from_index
   4.39%  libMESA_field_stat2.so  [.] has_buckets
   3.41%  libMESA_field_stat2.so  [.] hdr_value_at_index
   2.71%  libMESA_field_stat2.so  [.] hdr_next_non_equivalent_value
   2.38%  libMESA_field_stat2.so  [.] counts_get_normalised
   2.33%  libMESA_field_stat2.so  [.] hdr_median_equivalent_value
   2.08%  libMESA_field_stat2.so  [.] counts_get_direct
   1.88%  libMESA_field_stat2.so  [.] _update_iterated_values
   1.44%  libMESA_field_stat2.so  [.] _all_values_iter_next
   1.34%  libMESA_field_stat2.so  [.] normalize_index
   1.23%  libMESA_field_stat2.so  [.] highest_equivalent_value
   0.89%  libMESA_field_stat2.so  [.] hdr_iter_next
   0.46%  libMESA_field_stat2.so  [.] hdr_mean
   0.33%  libMESA_field_stat2.so  [.] hdr_value_at_percentile
   0.32%  [kernel]                [k] fget_light
   0.32%  [kernel]                [k] native_write_msr_safe
   0.28%  libc-2.17.so            [.] vfprintf
   0.23%  libc-2.17.so            [.] __select
   0.23%  libc-2.17.so            [.] __memset_sse2
   0.22%  tfe                     [.] ssl_cipher_apply_rule_constprop.5
```

Perf top



火焰图

小结：一些编码原则

重复会导致前后矛盾、产生隐微问题的代码，原因是当你修改重复点时，往往只改变了一部分而并非全部，这也意味着你对代码的组织没有想清楚。

Eric Raymond, The Art of UNIX Programming, 2003

Right Way vs Quick Way

Always choose right way. Technical Debt!

PREMATURE OPTIMIZATION IS THE ROOT OF ALL EVIL

过早优化是万恶之源

Donald Knuth, 1974

Debug

我复习的差不多了，稳了。。

然后考试↓↓↓

Program received SIGSEGV，Segmentation fault.

- 内存写越界
  - 通常在发生段错误临近的代码
  - 变量未初始化或初始化错误
  - 数据多线程访问冲突
  - 数组操作越界
  - 野指针
  - ……

代码检查方法

- 邻近代码分析
  - 通常段错误现场离其原因不远
  - 变量未初始化?
- 栈溢出检查
  - Linux的栈空间一般不超过16MB
  - 避免超大结构体造成栈溢出
- 内存拷贝越界
  - 检查内存申请的大小是否与memcpy/memset的一致
  - a=malloc(sizeof(struct info_t*));
  - memset(a,b,sizeof(struct info_t));
- 检查数组访问有无越界
  - 在访问数组时是否进行了边界检查
- 多线程的对共享数据的不安全访问
  - 输出运行日志

防御性编程

- 良好的编程风格
  - 输入参数用const修饰
- 在函数中增加断言，及早暴露问题
- 变量使用前初始化
- 指针释放后置NULL
  - 预防野指针
- 推迟变量的声明----讨论
  - 尽可能贴近使用的地方
- 审慎的进行类型强制转换
  - 强转是在告诉编译器：我比你懂得多，别做类型检查了
- 使用安全的数据结构
  - 避免str系列函数的使用

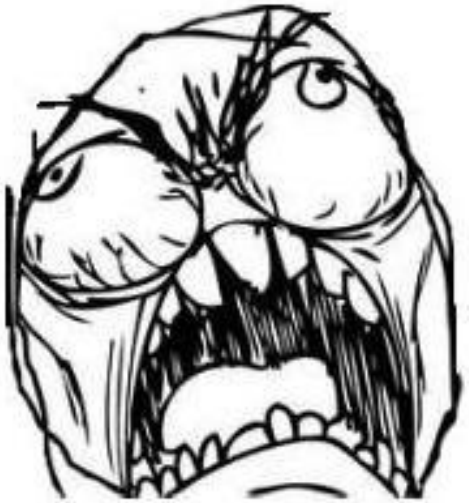
墨菲定律：凡是可能出错地方，一定会出错。

内存检查工具- C Programmer Only

- gdb
  - watch 内存地址，在内容改变时中断
- valgrind
  - Memcheck
  - Average slowdown ~10x
- AddressSanitizer
  - Average slowdown ~2x
  - https://github.com/google/sanitizers
- gperftools
  - Tcmalloc 内存池
  - Heap checker
  - https://github.com/gperftools/gperftools

难以调试的段错误

- 不能gdb
  - 没装gdb、编译没加-g、开了-O3、找不到源码。。。
- 出错在和用户代码无关的地方
  - fopen/malloc也能挂
- 段错误出现的时间不固定
  - 数小时/天挂一次
- 段错误的现场也不固定
  - 每个core文件都不一样
- Valgrind没问题/跑不动
  - 性能急剧下降
  - 只有大数据量才触发bug



它静静的等在那里，思考人生

- 多线程死锁
  - 少用、不用多线程
  - 代码中锁的获取和释放是成对出现的吗?
  - 多个锁在各处获得顺序相同吗?
  - gdb进去，查看锁的owner。
- 内存泄露
  - Valgrind、dictator2
- 线程混乱
  - 输出运行日志

还能怎么办？

- 要有信心：能可控重现的bug，都不是个事儿。
- 增加日志，保留现场。
- 离开座位，去喝杯咖啡。
- 找一个人，把代码讲给他听。
- 亡羊补牢，不如未雨绸缪。
  - 执行编码规范
  - 简化处理流程
  - 输出有效日志
- Debug难度
  - 死锁<内存泄漏<线程混乱<内存混乱

重构

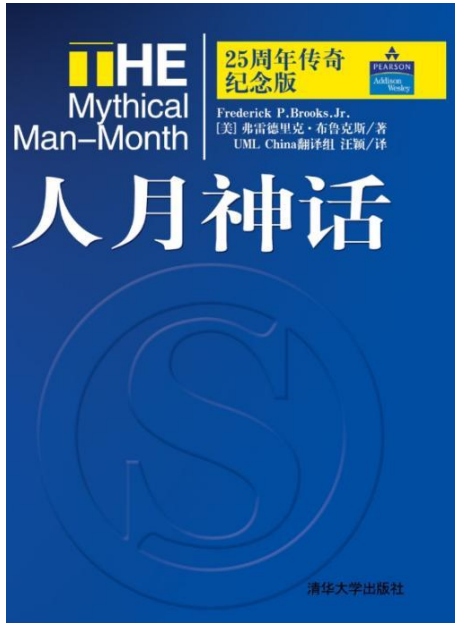
- 软件的“熵”
  - 熵是指某个系统中“无序”的总量
  - 热力学定律保证了宇宙中的熵倾向于最大化
  - 对于软件，“熵”就是复杂性
- 持续增加的复杂性，不断逼近管理的上限
  - 修复bug产生的补丁、例外的流程，蚕食原本整洁的代码
  - 直观的指标：if越来越多
- 通过重构，将“例外”变为“常规”
- 重构的技巧：
  - 积累单元测试用例
  - 经常小重构，大重构
  - 性能10倍，重新设计

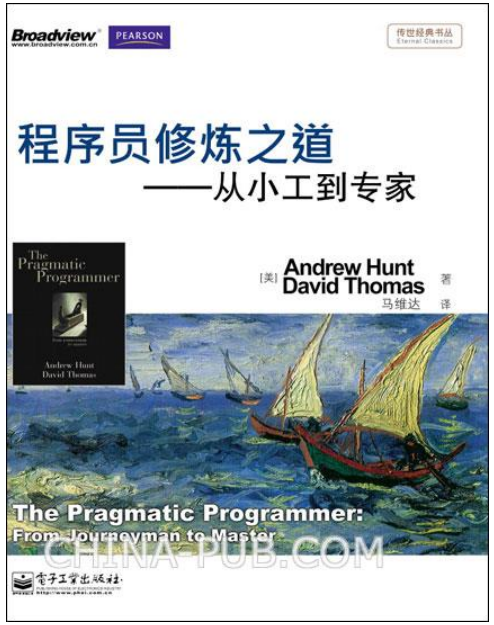
目录

- 建立好代码的品味
- 好代码始于得体的命名
- 分布式架构
- 编写代码
  - 程序->模块->函数->代码段
- 优化、调试与重构
- **不断的修炼**

推荐阅读

- 报告中的大部分内容可以在这些书籍中找到





不断的修炼

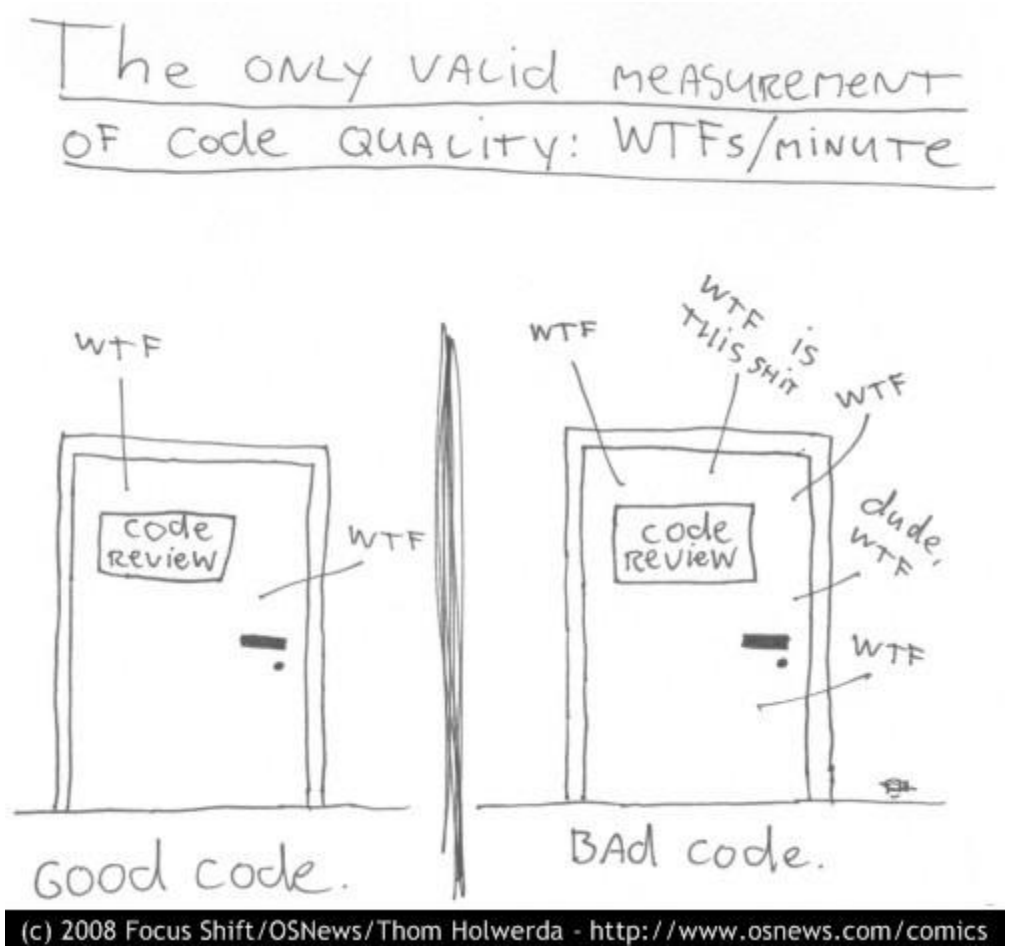
- 编写软件的历史已超过半个世纪，有很多经验可以被借鉴
  - 阅读优秀的开源代码
    - Libevent、Redis、UTHash
    - Linux kernel
  - 阅读书籍、参加外部会议
    - InfoQ、DPDK Summit、百度技术沙龙
  - 有hungry和foolish的感觉才会去学习
  - 最忌井底之蛙、夜郎自大
- 思考
  - 学而不思则罔
  - 不经过思考，不能形成自己的思想，等于白学和白干
- 面对自己的错误
  - 陈述事实，至少不要说谎
- 不要选择较容易的路
  - 因为容易走的路是下坡路。
- 享受创造的乐趣，为先进技术和精巧架构而颤栗。

Sign Your Work

- 传统上讲，我们都是手艺人，编程是我们谋生的技能。
- 也许将来就是你自己维护代码，如果你为了应付而coding，code只会应付你。
- 热爱编程的人，能够享受创造的喜悦。
- 在你的作品上，签上你的名字。

文森特·梵高

Code Review



代码质量的评价标准： WTF/分钟

你的感受

GEEDGE networks

RUN **FAST** | STAY **SECURE**

Guarantee Your Edge Network Security

网络边界的守卫者

积跬步 至千里